Taming the Stack: Proof-Preserving Blockwise FrodoKEM on RISC-V Devices with Hardware Acceleration

Frank Hartmann
{firstname}@hart.ma

November 27, 2025

Abstract

FrodoKEM provides conservative post-quantum security through unstructured lattices, yet its deployment on embedded systems is historically constrained by high memory requirements. While state-of-the-art implementations mitigate this by generating the public matrix on-the-fly, they remain bottlenecked by the sequential generation of secret matrices, which enforces a rigid trade-off between stack usage and recomputation overhead. To address this, we propose a blockwise secret generation mechanism that enables efficient random access to arbitrary matrix tiles. We formally prove that this modification is indistinguishable from the standard specification in the Random Oracle Model, thereby preserving IND-CCA2 security. By evaluating this approach on a 32-bit RISC-V core with custom cryptographic extensions, we demonstrate tiled encapsulation and key generation strategies that maintain constant transient stack usage (≈2–3 kB) across all parameter sets. This design effectively decouples memory footprint from algorithmic complexity, enabling flexible buffering strategies that optimize performance according to the available heap memory of the target platform.

1 Introduction

The rapid advancement of quantum computing has necessitated a global transition toward Post-Quantum Cryptography (PQC). While structured lattice schemes such as ML-KEM (Kyber) have been selected for standardization due to their efficiency, they rely on algebraic structures (e.g., module lattices) that may theoretically harbour unforeseen cryptanalytic vulnerabilities. As a hedge against this risk, FrodoKEM [1, 2] remains a critical alternative. Built upon the unstructured Learning With Errors (LWE) problem [3], it offers conservative security guarantees but demands significantly higher computational resources and memory bandwidth.

While implementing FrodoKEM on desktop computers is straightforward, doing so on resource-constrained embedded devices, such as ARM Cortex-M4 presents a distinct challenge. The scheme involves manipulating large matrices that far exceed the available SRAM of typical embedded platforms. To mitigate this, state-of-the-art implementations, such as those in pqm4 [4], generate the large public matrix on-the-fly in part. Further optimizations are possible by using the DSP that is present in some embedded devices to speed up the matrix multiplications or use a different PRNG to generate the large public matrix [5].

Another line of research implements FrodoKEM on FPGAs. Early FPGA implementations used a single multiply–accumulate (MAC) core, achieving modest throughput due to the heavy cost of generating the public matrix with SHAKE and sampling errors from a discrete distribution [6]. Later work showed that FrodoKEM can be significantly accelerated through parallel MAC architectures,

BRAM-based buffering, and careful dataflow design. Howe et al. [7] demonstrated 4×–16× parallel LWE multipliers, enabled by replacing SHAKE with Trivium to avoid bottlenecks in seed expansion, reaching up to 825–840 ops/sec while staying under 2000 slices . More recently, Düzyol et al. [8] proved that SHAKE itself is not a fundamental limitation, achieving 1 ms latency (≈1000 ops/sec) by scaling to 24× and 32× parallel multipliers using multi-BRAM architectures and double-buffered SHAKE generators—fully compliant with the official specification.

Additionally HW/SW-codesigns have been investigated. Banerjee et al. [9] first introduced Sapphire, a configurable crypto-processor that supported lattice-based schemes—including early FrodoKEM variants—by offloading SHAKE, sampling, and matrix operations to dedicated hardware units, demonstrating substantial speedups over pure software . Later in 2019, Dang et al. [10] extended the HW/SW-codesign perspective by benchmarking FrodoKEM on ARM–FPGA SoCs, showing how carefully chosen hardware accelerators (SHAKE, sampling, matrix multiplication) could yield performance gains of up to 28× for encapsulation and 20× for decapsulation, and detailing full accelerator architectures for FrodoKEM's bottlenecks . Building on these foundations, Karl et al. [11] presented the first tightly-coupled RISC-V crypto-coprocessor design tailored specifically to FrodoKEM, integrating Keccak-round accelerators, Gaussian-sampling logic, and custom MAC instructions directly into the ISA, achieving 7–8× speedups with minimal area overhead and demonstrating that FrodoKEM becomes practical for embedded devices when supported by lightweight, flexible on-core accelerators .

On plain microcontrollers, as analyzed by Bos et al. [12], implementing FrodoKEM introduces a challenging time–memory trade-off. Minimizing memory usage requires repeatedly regenerating portions of the public matrix on demand, whereas buffering larger tiles to improve performance quickly exhausts the limited SRAM available on low-end devices. This constraint fundamentally limits how aggressively one can optimize matrix operations. A significant—yet often overlooked—bottleneck within this trade-off is the generation of the *secret* matrices S and E. In the reference specification, both are derived from a seed via a monolithic extendable output function (XOF) stream. Because this stream is inherently sequential, accessing an arbitrary tile of the secret matrices during later phases of the computation (e.g., during matrix multiplications) requires either storing the entire matrices—which is prohibitive for higher parameter sets—or rewinding and regenerating the XOF output from the beginning. The latter incurs substantial latency and energy overhead.

Contributions In this work, we introduce a *blockwise* secret-generation mechanism for FrodoKEM which partly mitigates this bottleneck. By making the XOF input depend on explicit row and column indices—analogous to the strategy already used for the public matrix we enable efficient random access to arbitrary tiles of the secret matrices. We formally prove that this construction is indistinguishable from the original specification in the Random Oracle Model, thereby preserving IND-CCA2 security. We provide an extensive design space exploration to study the impact of our proposed change to the time–memory trade-off as well as a comprehensive comparison against other implementations.

We evaluate our approach on a 32-bit RISC-V core (CV32E40P) equipped with lightweight cryptographic extensions [11]. Blockwise secret generation enables "tiled" encapsulation and key-generation procedures that maintain constant transient stack usage across all FrodoKEM parameter sets. This effectively allows to tune time–memory strategies according to available space for accumulating results.

2 FrodoKEM

FrodoKEM [2] is a post-quantum key encapsulation mechanism built on the Learning With Errors (LWE) problem [3] in its "plain" matrix form. Its construction follows the standard paradigm of lattice-based public-key encryption: a large pseudorandom matrix serves as the public linear structure, while secrets and errors are drawn independently from a short discrete distribution. FrodoKEM's distinguishing feature is its avoidance of algebraic structure: all operations take place in $\mathbb{Z}_q$ without using rings or modules, thereby eliminating entire classes of structural cryptanalytic attacks.

Core Construction. Let $A \in \mathbb{Z}_q^{n \times n}$ be a public matrix derived from a short public seed seed_A using a XOF. The secret key contains a short matrix $S \in \mathbb{Z}_q^{n \times \bar{n}}$ and an error matrix $E \in \mathbb{Z}_q^{n \times \bar{n}}$, both sampled entry-wise from a discrete noise distribution χ. The public key is defined as:

$$B = AS + E \in \mathbb{Z}_q^{n \times \bar{n}}.$$

This forms an instance of the LWE problem: B is computationally indistinguishable from a uniformly random matrix unless an adversary can recover S from (A, B).

Encapsulation. To encapsulate a key, the sender samples fresh "ephemeral" matrices $S' \in \mathbb{Z}_q^{\bar{m} \times n}$, $E' \in \mathbb{Z}_q^{\bar{m} \times n}$, and $E'' \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$, again from χ. Using these, the ciphertext is formed as

$$B' = S'A + E', \qquad C = S'B + E'' + \mathrm{Encode}(\mu),$$

where μ is a message-derived bitstring used within the Fujisaki–Okamoto (FO) transform [13]. The pair (B', C) constitutes the ciphertext. The shared secret is then derived by hashing (μ, B', C).

Decapsulation. To decapsulate, the recipient uses the secret matrix S to remove the LWE error from C and recover μ. From μ and the ciphertext components, the same key-derivation hash is applied to reconstruct the shared secret. The FO transform enforces correctness and CCA security: if the recomputed ciphertext does not match the input ciphertext, the output key is replaced with a pseudorandom fallback value, preventing misuse of decryption failures.

Pseudorandomness FrodoKEM relies on expanding short seeds into large pseudorandom structures—most importantly the public matrix A and the secret and error matrices used throughout key generation, encapsulation, and decapsulation. Because these objects contain thousands of entries, the scheme defines deterministic expansion mechanisms ensuring high entropy and reproducibility. The recommended method uses SHAKE, built on the Keccak sponge construction. In the Keccak sponge the 1600-bit state is organized as a 5×5 array of 64-bit lanes. During *absorption*, the input seed is XORed into the first r bits of the state (the "rate"), with each absorbed block followed by the Keccak-f[1600] permutation. After absorption, the sponge enters the *squeezing* phase, where repeated permutations yield arbitrarily long pseudorandom output.

As an alternative, AES-CTR expands a seed used as an AES key by encrypting successive counter blocks. In the following we will restrict the analysis to the SHAKE-based version.

Technical specifications. The FrodoKEM family consists of three parameter sets corresponding to increasing classical and quantum security levels. Each variant is defined by the matrix dimension n, the modulus q, and the discrete noise distribution χ, which is parameterized by a standard deviation σ and a finite support interval. Larger parameter sets use larger n and higher moduli to increase the hardness of the underlying LWE problem, while simultaneously reducing the noise magnitude to maintain correctness during decryption. All variants use the same secret and error matrix shape $\bar{m} \times \bar{n} = 8 \times 8$, ensuring that the high-level structure of the scheme remains identical across parameter sets. The specified parameters are given in Table 1.

Scheme	n	q	σ	support of χ	$\bar{m} \times \bar{n}$
FrodoKEM-640	640	2^{15}	2.8	$[-12 \dots 12]$	8×8
FrodoKEM-976	976	2^{16}	2.3	$[-10 \dots 10]$	8×8
FrodoKEM-1344	1344	2^{16}	1.4	$[-6 \dots 6]$	8×8

Table 1: Parameters of the FrodoKEM parameter sets.

Security. The security of FrodoKEM reduces tightly to the hardness of the LWE problem with parameters (n, q, χ) and the soundness of the Fujisaki–Okamoto transform [13]. All randomness in the scheme is derived from a XOF, which is modeled as a random oracle in the security proof. Since the construction uses no algebraic structure beyond linear algebra over $\mathbb{Z}_q$, its security does not rely on assumptions about number-theoretic rings or modules.

3 FrodoKEM on Embedded Devices

The pqm4 framework [4] provides highly optimized implementations of post-quantum schemes for the ARM Cortex-M4 microcontroller. In this work, we rely on its implementation of FrodoKEM using SHAKE-based expanders. Since FrodoKEM is a lattice-based KEM of the "plain" LWE family, its computational profile on an embedded platform is dominated by three components: (i) expansion of randomness via SHAKE, (ii) sampling of noise from a discrete distribution, and (iii) structured matrix–vector multiplications over $\mathbb{Z}_q$.

On-the-fly generation of the public matrix A key characteristic of the pqm4 implementation is that only up to 8 rows of the public matrix A are present at any time in the RAM. Using these rows the microcontroller computes the required partial products. This strategy drastically reduces static memory requirements and shifts the computational cost to the computations of SHAKE, which is computed with tailored assembly routines.

Noise sampling and secret structure All secret and error matrices in FrodoKEM are sampled from a discrete noise distribution specified by the scheme. On pqm4, this is implemented by expanding a short seed with SHAKE and transforming the output into noise values using a small sampler based on a Look Up Table. Because noise generation is invoked several times—in KeyGen, encapsulation, and decapsulation—its efficiency is crucial to overall performance.

Matrix multiplications on Cortex-M4 The most expensive arithmetic in FrodoKEM consists of multiplying large rectangular matrices modulo q. On the Cortex-M4, these operations are implemented as tightly optimized loop nests that exploit the processor's DSP instructions, such as

16-bit multiply–accumulate operations. Since A is generated in partly, the matrix multiplications align with these chunks: computations are performed as row-times-column (KeyGen) or row-by-chunk (Encaps/Decaps), and the temporary memory footprint is kept minimal. This approach achieves a balance between computational efficiency and stack usage.

4 Accelerating FrodoKEM on RISC-V with Custom Instructions

This SHAKE-centric behavior becomes even more pronounced on 32-bit RISC-V microcontrollers. FrodoKEM's performance on these systems is dominated by three computational kernels. Firstly, the Keccak-f[1600] permutation incurs overhead due to its 64-bit lanes. Secondly, matrix multiplications require thousands of 16-bit products. Lastly, Gaussian sampling must be performed for every secret coefficient. Consequently, these kernel dictate the overall runtime of defined operations. This continuity in bottlenecks across platforms naturally motivates architecture-level optimizations that integrate lightweight accelerators directly into the pipeline of small RISC-V cores.

Custom RISC-V Support for FrodoKEM. To address this, Karl et al. [11] introduced lightweight custom instructions for the CV32E40P core, whose compact pipeline and flexible decoder make it well-suited for targeted cryptographic extensions. Their acceleration strategy augments the base RISC-V ISA with three families of Frodo-oriented custom instructions:

1. **Keccak-f[1600] round instruction `keccak.f1600` :** A tightly integrated permutation instruction enables single-cycle updates to the 1600-bit state.
2. **Packed 16-bit multiply–accumulate `pq.mac2` :** This instruction performs two parallel halfword multiplications with accumulation, substantially improving the throughput of Frodo's inner-product–dominated matrix multiplications.
3. **Dedicated Gaussian sampling instructions `pq.frodo_xxx` :** These instructions implement minimized combinational circuits that convert uniform input into discrete Gaussian samples with constant latency.

When combined, the custom extensions provide 7–8× speedups for key generation, encapsulation, and decapsulation on CV32E40P-class microcontrollers. These results show that FrodoKEM—often considered too heavy for embedded devices—can be made practical on small RISC-V platforms with tightly coupled hardware accelerators.

5 Modifying FrodoKEM for Embedded Devices

From the previous sections it should be apparent the original FrodoKEM specification [2] is not ideally suited for constrained embedded environments. Therefore prior art already proposed changes such as using a different XOF i.e. Trivium. However this is not sufficient in all cases and therefore changes to the *algorithms themselves* have been proposed.

In [12] Bos et al. propose a modified matrix-generation routine that divides each row of A into tiles matching the SHAKE128 rate, as shown in Fig. 1. Instead of generating an entire row at once, each row i is split into blocks of $w = 84$ entries. For each tile, the generator absorbs both the row index i and a *column index* j into SHAKE128:

$$b = \langle i \rangle \,\|\, \langle j \rangle \,\|\, \mathsf{seed}_A.$$

A single SHAKE call then produces exactly 84 16-bit values, i.e. 168 bytes, corresponding to one rate-sized tile. By introducing the column index j, the generator can produce arbitrary tiles of A independently and in any order. This eliminates the need to store entire rows or the entire matrix, enabling implementations that keep only a single tile in memory at all times.

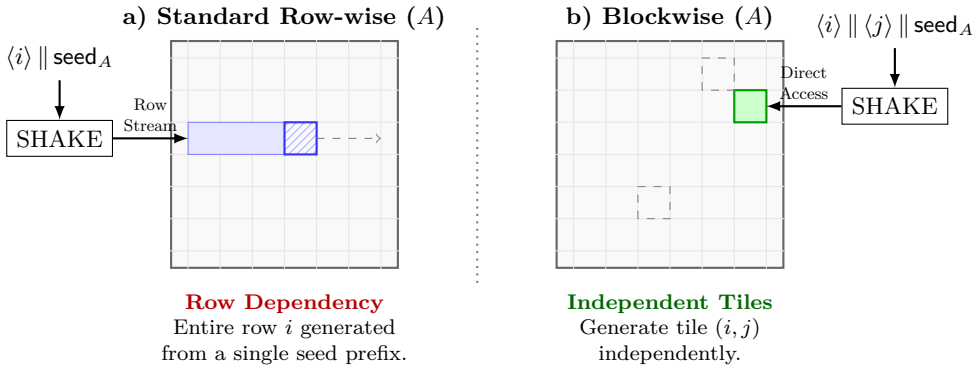


Figure 1: Visual comparison of public matrix A generation. **Left:** The standard specification generates a full row from the row index i. **Right:** The tiled approach [12] absorbs both row i and column j, enabling random access to sub-blocks.

6 Time–Memory Trade-Off

The tiled SHAKE generation of A leads directly to an explicit time–memory trade-off, parameterized by t_k and t_{ed}. Following Bos et al., t_k denotes the number of rows of A that are buffered simultaneously during KeyGen, while t_{ed} denotes the number of additional rows of S' buffered in Encaps and Decaps. Small values of t_k and t_{ed} minimize the number of SHAKE-generated tiles that must be stored at once, but force more frequent re-computations. In KeyGen S must recomputed n/t_k times. In Enc and Dec A must be recomputed $n/(t_{\text{ed}} + 1)$ times. Larger values increase the number of tiles available in RAM, thereby reducing the number of SHAKE128 calls and improving runtime, but at the cost of a higher stack footprint.

Table 2 summarizes this behavior in concrete terms for FrodoKEM-640. In KeyGen, the memory for A scales as $168 \cdot t_k$ bytes, while the memory for B and E scales as $16 \cdot t_k + 200$ bytes once on-the-fly packing and SHAKE state reuse are taken into account. For Encaps and Decaps, the memory for A becomes $168 \cdot (t_{\text{ed}} + 1)$ bytes, and the dominant terms in the storage for B' and S' are linear in t_{ed} (e.g., $1{,}280 \cdot t_{\text{ed}}$ or $1{,}280\,(t_{\text{ed}} + 1)$ bytes). In the limiting case $t_k = 8$ and $t_{\text{ed}} = 7$, the implementation essentially behaves like the original SHAKE reference code with maximal performance and memory usage. For $t_k = 1$ and $t_{\text{ed}} = 0$, only a single SHAKE tile fits in RAM, giving minimal stack usage but a substantially increased cycle count due to intensive recomputation of SHAKE-generated tiles.

The SHAKE-based trade-off implementation significantly reduces stack consumption, but it also introduces several performance-related implications. Most notably, the parameters t_k and t_{ed} create an inherent tension between memory usage and runtime: achieving low cycle counts requires larger buffers to hold more SHAKE-generated tiles, while smaller buffers force the implementation to repeatedly regenerate matrix blocks. Since both A and the secret matrices S and S' are produced via SHAKE128, reduced values of t_k and t_{ed} increase the total number of Keccak-f permutations, which

FrodoKEM-640.KeyGen

Implementation	A	B, E	S
pqm4	5120	10,240	10,240
Bos et al.	$168 \cdot t_k$	$16 \cdot t_k + 200$	$20 + 200$

FrodoKEM-640.Encaps

Impl.	A	B	V, E"	C	B′, E′	S′
pqm4	10,240	10,240	128	128	10,240	10,240
Bos	$168 \cdot (t_{\text{ed}} + 1)$	128	128	128	$16 + 1,280 \cdot t_{\text{ed}}$	$1,280\,(t_{\text{ed}} + 1) + 1,664$

FrodoKEM-640.Decaps

Impl.	A	B	B′	S	M, V, E"	C	C′	B″, E′	S′
pqm4	10,240	10,240	10,240	10,240	128	128	128	10,240	10,240
Bos	$168(t_{\text{ed}} + 1)$	128	128	128	128	128	128	$16 + 1,280 t_{\text{ed}}$	$1,280\,(t_{\text{ed}} + 1) + 1,664$

Table 2: Memory breakdown for FrodoKEM-640 (SHAKE) using the tiled SHAKE matrix generation of A vs. standard pqm4.

dominate runtime in low-memory configurations. This effect is exacerbated by the slow incremental SHAKE API, whose overhead becomes substantial when tiles must be regenerated many times.

7 Technical Overview of Blockwise Secret Matrix Generation

This motivates a more fundamental redesign of how FrodoKEM generates its secret matrices. Our central insight is that all three drawbacks identified above can be addressed by altering the way the secret matrices S and S' are produced. Inspired by the blockwise generation of the public matrix A by Bos et al., we replace the traditional monolithic SHAKE expansion with a structured, indexed approach. Instead of deriving the entire matrix from a single XOF stream, we introduce explicit row and column indices and generate matrix tiles independently, as illustrated in Fig. 2. The full procedure is specified in Algorithm 1.

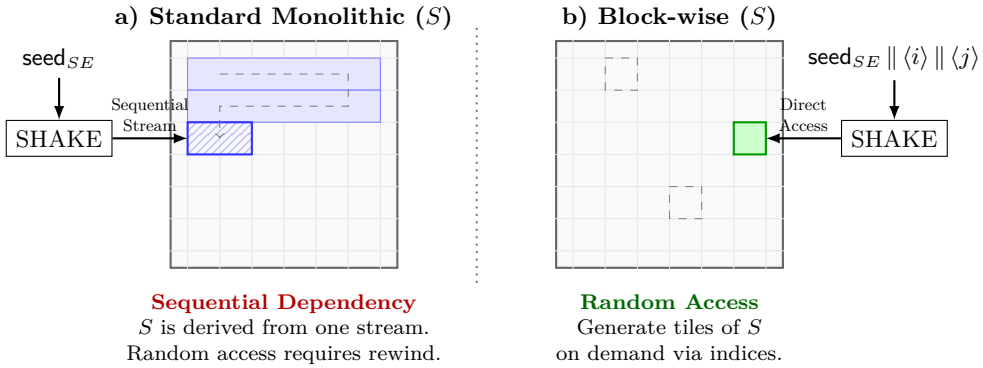


Figure 2: Visual comparison of secret matrix generation strategies. **Left:** The standard specification derives S from a single seed sequentially. **Right:** The proposed block-wise approach uses indices (i, j) to feed SHAKE, allowing independent generation of specific tiles.

Algorithm 1 Proposed generation of S'

Input: $seed_{SE}$
Output: Matrix S'
$n' \leftarrow 84 \cdot \lfloor \frac{n}{84} \rfloor$
$n'' \leftarrow n \bmod 84$
for $(i = 0; i < \bar{n}; i \leftarrow i + 1)$ **do**
    for $(j = 0; j < n'; j \leftarrow j + 1)$ **do**
        $b \leftarrow seed_{SE} \| \langle i \rangle \| \langle j \rangle$ where $\langle i \rangle, \langle j \rangle \in \{0,1\}^8$
        $S'[i \cdot n + j \cdot 84] \leftarrow \text{SHAKE128}(b, 16 \cdot 84)$
    end for
    $b \leftarrow seed_{SE} \| \langle i \rangle \| \langle n' \rangle$ where $\langle i \rangle, \langle n' \rangle \in \{0,1\}^8$
    $S'[i \cdot n + n'] \leftarrow \text{SHAKE128}(b, 16 \cdot n'')$
end for
return S'

Generation of E, E' and E'' For completeness we want to mention that the proposed change necessitates to specify how the error matrices E, E' and E'' are generated. In our case we simply chose to generate these matrices monolithically based on $seed_{SE}$ and a distinct set of indices i, j for each matrix that is not used for the generation of S respectively S'.

Limitation We fix the block size to $r_{\text{SHAKE128}} = 168$ bytes for all FrodoKEM parameter sets. Although FrodoKEM-976 and FrodoKEM-1344 rely on SHAKE256 for generating S and S', and therefore require more Keccak permutations under this design, the uniform block size enables simple and efficient multiplication routines. The resolution of using SHAKE256 as originally specified is left open for further research.

8 Security Proof: IND-CCA2 Reduction via Game Hopping and Tightness

Before we delve into how this change is beneficial for implementation we will formally analyze the security of the proposed blockwise FrodoKEM-640 variant. Our objective is to demonstrate that the scheme achieves IND-CCA2 security in the Random Oracle Model (ROM). We show that no probabilistic polynomial-time (PPT) adversary can distinguish the real shared secret from a random bit string with non-negligible advantage, even when granted adaptive access to a decapsulation oracle. The proof employs a sequence of games (Game Hopping) to reduce the security of the blockwise scheme to that of the standard FrodoKEM-640 scheme, and subsequently to the hardness of the Learning With Errors (LWE) problem via the Fujisaki–Okamoto (FO) transform [13, 2].

8.1 Preliminaries and Distributional Analysis

We first establish that the blockwise generation of secret matrices is stochastically equivalent to the monolithic generation used in the standard specification.

Lemma 1 (Distribution Equivalence in ROM)**.** *Let $\mathcal{G}_{block}$ denote the blockwise secret generation algorithm (Algorithm 1) and $\mathcal{G}_{std}$ denote the standard monolithic generation algorithm. Let $\mathcal{O} : \{0,1\}^* \to \{0,1\}^\infty$ be a random oracle instantiated by* SHAKE*. For any fixed high-entropy seed*

$\mathsf{seed}_{SE} \in \{0,1\}^\kappa$, *let* $(S_{block}, E_{block}) \leftarrow \mathcal{G}^{\mathcal{O}}_{block}(\mathsf{seed}_{SE})$ *and* $(S_{std}, E_{std}) \leftarrow \mathcal{G}^{\mathcal{O}}_{std}(\mathsf{seed}_{SE})$. *Then, the statistical distance between the joint distributions of the outputs is zero:*

$$\Delta\left((S_{block}, E_{block}),\ (S_{std}, E_{std})\right) = 0. \tag{1}$$

Proof. The standard algorithm $\mathcal{G}_{\text{std}}$ queries the random oracle $\mathcal{O}$ on input seed_{SE} to produce a monolithic stream $Y \in \{0,1\}^L$. The blockwise algorithm $\mathcal{G}_{\text{block}}$ performs a sequence of queries on distinct inputs $x_{i,j} = \mathsf{seed}_{SE} \,\|\, \langle i \rangle \,\|\, \langle j \rangle$, producing outputs $y_{i,j}$. In the Random Oracle Model, $\mathcal{O}(x)$ outputs uniformly random bits for any distinct query x. Since the domain separation indices (i, j) ensure that all inputs $x_{i,j}$ are unique and distinct from the monolithic input (assuming proper domain separation tags or length prefixing as per SHAKE standards), the concatenation of block outputs $Y' = \Big\|_{i,j} y_{i,j}$ is distributed uniformly and independently, identical to Y. Consequently, applying the deterministic sampler χ to Y and Y' yields identical distributions for the entries of S and E. □

8.2 IND-CCA2 Security Reduction

We now prove that the blockwise modifications do not compromise the chosen-ciphertext security of the KEM.

Theorem 1 (IND-CCA2 Security of Blockwise FrodoKEM-640)**.** *Let* PKE *be the underlying IND-CPA secure public-key encryption scheme of* FrodoKEM*. In the Random Oracle Model, the blockwise* FrodoKEM-*640 Key Encapsulation Mechanism (*KEM_{block}*) is IND-CCA2 secure assuming the hardness of the* $\text{LWE}_{n,q,\chi}$ *problem. Formally, for any PPT adversary* $\mathcal{A}$ *attacking* KEM_{block} *with at most* q_{RO} *random oracle queries and* q_D *decapsulation queries, there exist efficient reduction algorithms* $\mathcal{B}$ *and* $\mathcal{C}$ *such that:*

$$\text{Adv}^{\text{IND-CCA2}}_{\mathsf{KEM}_{block}}(\mathcal{A}) \ \le\ \text{Adv}^{\text{IND-CPA}}_{\mathsf{PKE}}(\mathcal{B}) \ +\ \text{Adv}_{\text{LWE}}(\mathcal{C}) \ +\ \frac{q_{RO}}{2^\kappa}, \tag{2}$$

where κ *is the seed length. The reduction is tight.*

Proof. We define a sequence of games $G_0, \ldots, G_3$. Let S_i be the event that the adversary wins in Game G_i.

Game G_0 **(Real Experiment).** This is the standard IND-CCA2 security experiment for $\mathsf{KEM}_{\text{block}}$.

1. The challenger generates a key pair (pk, sk) using the blockwise generator $\mathcal{G}_{\text{block}}$.
2. The challenger samples a random bit $b \leftarrow \{0,1\}$. If $b = 0$, the challenger returns a real key K^* and ciphertext C^*; if $b = 1$, it returns a random key $K \leftarrow \mathcal{K}$ and C^*.
3. The adversary $\mathcal{A}$ has access to a decapsulation oracle $\mathcal{O}_{Dec}(\cdot)$ which computes decapsulation using the blockwise regeneration of secrets. $\mathcal{A}$ may not query $\mathcal{O}_{Dec}(C^*)$.
4. $\mathcal{A}$ outputs a guess b'.

By definition, $\text{Adv}^{\text{IND-CCA2}}_{\mathsf{KEM}_{\text{block}}}(\mathcal{A}) = |\Pr[S_0] - 1/2|$.

Game G_1 **(Transition to Monolithic Generation).** In this game, we modify the challenger's internal algorithms (KeyGen, Encaps, Decaps). Whenever the challenger is required to generate secret matrices S, S' or errors E, E', E'', it employs the standard monolithic generator $\mathcal{G}_{\text{std}}$ instead of $\mathcal{G}_{\text{block}}$. By Lemma 1, the distributions of the keys, ciphertexts, and shared secrets are identical in the ROM. The adversary's view remains unchanged. Thus, $\Pr[S_1] = \Pr[S_0]$.

Game G_2 (Standard FrodoKEM-640 Environment). We observe that Game G_1 is functionally equivalent to the IND-CCA2 experiment for the *standard* FrodoKEM-640 scheme, denoted as $\mathsf{KEM}_{\text{std}}$. The inputs, outputs, and internal states match the reference specification exactly. This step is purely syntactic. We are now analyzing the security of the standard scheme. Thus, $\Pr[S_2] = \Pr[S_1]$.

Game G_3 (Reduction to Hardness Assumptions). In this game, we replace the interaction with the $\mathsf{KEM}_{\text{std}}$ challenger with a reduction to the underlying cryptographic problems. According to the security proofs provided in the original FrodoKEM specification [2] and the generic properties of the FO transform [13], the advantage of an adversary against $\mathsf{KEM}_{\text{std}}$ is bounded by the advantage against the IND-CPA security of the underlying PKE scheme (which relies on LWE) plus negligible terms related to the ROM.

$$|\Pr[S_2] - 1/2| \leq \mathrm{Adv}^{\text{IND-CPA}}_{\mathsf{PKE}}(\mathcal{B}) + \mathrm{Adv}_{\text{LWE}}(\mathcal{C}) + \mathrm{negl}(\lambda). \tag{3}$$

Combining the bounds. Combining the transitions, we obtain:

$$\mathrm{Adv}^{\text{IND-CCA2}}_{\mathsf{KEM}_{\text{block}}}(\mathcal{A}) = |\Pr[S_0] - 1/2| = |\Pr[S_1] - 1/2| = |\Pr[S_2] - 1/2|.$$

Substituting the bound from Game G_3, we arrive at inequality (2). □

8.3 Tightness

The reduction entails a single transformation step ($G_0 \to G_1$) distinct from the standard FrodoKEM proof. Since this step relies on the Distribution Equivalence Lemma, which establishes perfect distributional identity in the Random Oracle Model, it introduces zero adversarial advantage loss. The reduction tightness is therefore identical to that of the standard FrodoKEM scheme; the only security loss factors are those inherent to the Fujisaki–Okamoto transform and the LWE reduction itself. Thus, the blockwise variant preserves the conservative security guarantees of the original design.

9 Design Space Exploration

The implementation of post-quantum cryptography on constrained embedded devices requires a careful analysis of the trade-offs between memory consumption, computational cycles, and I/O overhead. A significant design decision remains regarding the buffering of the result matrices.

9.1 Key Generation Buffering Strategies

In KeyGen, the device must compute $B = AS + E$. Because A and S can be generated tile-wise on demand, the limiting factor for RAM usage becomes the storage of the output matrix B. We identify three distinct strategies for managing this accumulation.

Trade-Off Formalization We introduce the following trade-off parameters:

- t_k: This denotes the number of row-blocks of A that are processed while a single block of S is held in memory. Consequently, the entire matrix S must be regenerated $\lceil n/t_k \rceil$ times.
- t_c: This denotes the number of column-blocks of S that are buffered in parallel. Consequently, the required rows of A must be regenerated $\lceil \bar{n}/t_c \rceil$ times.

Strategy A: Direct Blockwise Streaming. The device computes a single block of B, immediately packs and hashes it into the public key stream, and discards it before starting the next block.

- **Mechanism:** We restructure the computation to finalize a specific output block of B entirely before moving to the next. This requires iterating through the full inner dimension, accumulating the product of the corresponding generated blocks of A and S into a small temporary buffer.
- **RAM:** We allocate buffers for only the *currently active* blocks of A, S, and the accumulation block for B. Thus, we have a minimal main RAM requirement of approximately 3×168 bytes.
- **Recomputation Effort:** Since no parts of A or S are cached for reuse across different output blocks, we are effectively forced to set $t_k = 1$ and $t_c = 1$. Consequently, S needs to be recomputed n times (once for every row of A) and the relevant rows of A need to be recomputed $\bar{n}$ times (once for every column of B).

Strategy B: Windowed Buffering. The device allocates a temporary "window" buffer for k consecutive blocks of B, fills it completely via matrix multiplication, and then processes (packs/hashes) the window in bulk before reusing the memory.

- **Mechanism:** We compute B in batches of k consecutive blocks. This allows us to perform the packing and hashing operations on larger chunks of data.
- **RAM:** We allocate buffers for one block of A and one block of S for the current tile, additionally k blocks of B. The RAM requirement scales linearly with approx. $k \times 168$ bytes.
- **Recomputation Effort:** By buffering k blocks, we effectively set $t_k \approx k$, reducing the recomputation of S to roughly n/k times. However, typically t_c remains 1, meaning A must still be recomputed $\bar{n}$ times.

Strategy C: Full Result Buffering. The device allocates a contiguous buffer for the entire matrix B and accumulates partial products into this memory as they are computed.

- **Mechanism:** We exploit the fact that A and S consist of blocks of matching size. Thus, we can compute the row-column products by only buffering matching blocks of these matrices.
- **RAM:** We allocate one buffer of size $2n\bar{n}$ for B. Additionally, we allocate buffers for only one block of A and one block for each column of S on the stack, giving ≈ 1.5 kB of additional RAM requirement.
- **Recomputation Effort:** Since B is available in full, we can iterate optimally. We set $t_k = n$ (implying S is computed once) and effectively set $t_c = \bar{n}$ (implying A is computed once). Thus, there is no recomputation at all.

Table 3: Comparison of buffering strategies, recomputation overhead, and memory usage.

Strategy	t_k	t_c	Recomp. S	Recomp. A	RAM Req.
A: Streaming	1	1	n	$\bar{n}$	Minimal (≈ 504 bytes)
B: Windowed	k	1	n/k	$\bar{n}$	Tunable ($\propto k$)
C: Full Result	n	$\bar{n}$	1	1	High ($\propto n$)

Summary of KeyGen Improvements. The proposed blockwise generation of S fundamentally alters the memory landscape of the key generation phase. By enabling random access to individual tiles of the secret matrix, we eliminate the dependency between stack consumption and the performance parameter t_k observed in previous works. The choice of buffering strategy becomes purely a function of available memory for partial accumulation of results rather than a complex optimization of stack usage versus regeneration cost. Consequently, the memory bottleneck shifts entirely from the inputs (A, S) to the output (B), granting the system designer fine-grained control over the time–memory trade-off.

9.2 Encapsulation Buffering Strategies

In the encapsulation phase (Enc), the device must compute two LWE samples involving the ephemeral secret matrix $S' \in \mathbb{Z}_q^{\bar{m} \times n}$: $B' = S'A + E' \in \mathbb{Z}_q^{\bar{m} \times n}$ and $V = S'B + E'' \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$. We identify three distinct strategies for managing the accumulation of B'.

Formalization of Tiled Recomputation. We describe the execution grid of the matrix multiplication $B' = S'A$ using three orthogonal tiling parameters:

- t_{rp} (vertical tiling), the number of rows of B' computed in parallel
- t_{ip} (inner tiling), the number of inner-dimension blocks processed per accumulation step
- t_{ed} (horizontal tiling), the number of output column blocks computed in parallel

General Performance Formulas The formulas are different for each strategy. Let r_{block} denote the number of q-elements output per SHAKE128 block and define $N_{\text{blks}} = \lceil n / r_{\text{block}} \rceil$.

Strategy A: Fully Stack-Resident Tiled Processing

All working sets—tiles of A, S', and partial accumulators for B'—are stored entirely on the stack. This forces very small tiling factors and yields the greatest recomputation overhead.

- **Mechanism.** The kernel processes t_{rp} rows of S' and t_{ed} column blocks of A concurrently, with $t_{\text{ip}} = 1$ inner blocks per step. After each horizontal batch, the produced slice of B' is written out immediately before proceeding.
- **Parameter Selection.** Stack constraints typically require $t_{\text{rp}} \in \{1, 2\}$, $t_{\text{ip}} = 1$ and $t_{\text{ed}} \in \{1, 2\}$.
- **Memory.** Peak stack usage is approximately $M_{\text{stack}} \approx t_{\text{rp}} t_{\text{ip}} r_{\text{SHAKE128}} + t_{\text{ed}} t_{\text{rp}} r_{\text{SHAKE128}} +$ overhead, growing linearly in both t_{rp} and t_{ed}.

- **Recomputation Effort.** The matrix A is regenerated for every horizontal batch, so $N_{\text{recomp}}(A) = \lceil N_{\text{blks}}/t_{\text{ed}} \rceil$. The matrix S' is regenerated for every vertical pass and every horizontal batch, giving $N_{\text{recomp}}(S') = \lceil \bar{m}/t_{\text{rp}} \rceil \cdot \lceil N_{\text{blks}}/t_{\text{ed}} \rceil$.

Strategy B: Windowed Buffering

Strategy B allocates a sliding window of the output B' and additional space for tiles of A and S'.

- **Mechanism.** The output B' is divided into horizontal windows of width w blocks. A small heap buffer holds one such window. Once window j is completed, it is emitted and the buffer is reused for window $j + 1$.
- **Parameter Selection.** Stack constraints typically require $t_{\text{rp}} \in \{2, 4\}$, $t_{\text{ip}} = 1$, $t_{\text{ed}} \in \{1, 2, 4\}$, with the window width bounded by $t_{\text{ed}} \leq w \leq \min\{N_{\text{blks}}, W_{\max}\}$.
- **Memory.** The stack usage is approximately $M_{\text{stack}} \approx t_{\text{rp}} t_{\text{ip}} r_{\text{SHAKE128}} + t_{\text{ed}} t_{\text{rp}} r_{\text{SHAKE128}} + t_{\text{rp}} w r_{\text{SHAKE128}}$.
- **Recomputation Effort.** The matrix A is regenerated once per window, so $N_{\text{recomp}}(A) = \lceil N_{\text{blks}}/w \rceil$. The matrix S' is regenerated once per vertical pass and once per window, giving $N_{\text{recomp}}(S') = \lceil \bar{m}/t_{\text{rp}} \rceil \cdot \lceil N_{\text{blks}}/w \rceil$.

Strategy C: Heap-Backed Blockwise Processing

Strategy C stores the entire output matrix B' on the heap, eliminating horizontal stack constraints and enabling fully optimized tiling.

- **Mechanism.** A fixed-register kernel uses $t_{\text{rp}} = 4$ and $t_{\text{ip}} = 2$. The horizontal tiling t_{ed} becomes a free performance parameter because B' is fully heap-resident. Only a constant-size working set of A and S' resides on the stack.
- **Parameter Selection.** Stack constraints typically require $t_{\text{ed}} \in \{1, 2, \lceil N_{\text{blks}}/2 \rceil, N_{\text{blks}}\}$.
- **Memory.** Stack usage is essentially constant and can be approximated as $M_{\text{stack}} \approx (t_{\text{rp}} t_{\text{ip}} + 2t_{\text{ip}}) r_{\text{SHAKE128}} \approx 2\,\text{kB}$.
- **Recomputation Effort.** The matrix A is regenerated exactly once per block, so $N_{\text{recomp}}(A) = N_{\text{blks}}$. The matrix S' is regenerated for each vertical pass and each horizontal batch: $N_{\text{recomp}}(S') = \lceil \bar{m}/t_{\text{rp}} \rceil \cdot \lceil N_{\text{blks}}/t_{\text{ed}} \rceil$. For $\bar{m} = 8$ and $t_{\text{rp}} = 4$, this simplifies to $N_{\text{recomp}}(S') = 2 \cdot \lceil N_{\text{blks}}/t_{\text{ed}} \rceil$.

Summary of Encapsulation Improvements. The tiled recomputation framework decouples the dominant cost in encapsulation, namely the evaluation of $B' = S'A$, from the strict stack limitations of embedded targets. By expressing the computation in terms of independent vertical, inner, and horizontal tiles, we obtain three practical buffering strategies that trade re-computation of S' and A against stack usage. As a result, the memory bottleneck shifts entirely to the output buffer for B', and encapsulation becomes amenable to fine-grained tuning.

Table 4: Comparison of encapsulation buffering strategies, recomputation overhead, and memory usage.

Strategy	t_{rp}	t_{ed}	Recomp. S'	Recomp. A	Memory Usage
A:	1–2	1–2	$\left\lceil \frac{\bar{m}}{t_{\mathrm{rp}}} \right\rceil \left\lceil \frac{N_{\mathrm{blks}}}{t_{\mathrm{ed}}} \right\rceil$	$\left\lceil \frac{N_{\mathrm{blks}}}{t_{\mathrm{ed}}} \right\rceil$	Minimal stack only
B:	2–4	1–4	$\left\lceil \frac{\bar{m}}{t_{\mathrm{rp}}} \right\rceil \left\lceil \frac{N_{\mathrm{blks}}}{w} \right\rceil$	$\left\lceil \frac{N_{\mathrm{blks}}}{w} \right\rceil$	Tunable window ($\propto w$)
C:	4	1–N_{blks}	$\left\lceil \frac{\bar{m}}{t_{\mathrm{rp}}} \right\rceil \left\lceil \frac{N_{\mathrm{blks}}}{t_{\mathrm{ed}}} \right\rceil$	N_{blks}	High ($\propto n$)

10 Implementation on RISC-V with Custom Instructions

To assess the practicality of the blockwise variant, we implemented the full FrodoKEM scheme on a resource-constrained RISC-V platform. Our target is the CV32E40P, a 32-bit core developed within the PULP ecosystem and integrated with tightly coupled accelerator modules [11], as detailed in Section 4. All three KEM operations—KeyGen, Encaps, and Decaps—use *Strategy C*, the fully blockwise matrix-generation approach.

10.1 Stack Usage and Memory Layout

Using Strategy C uniformly across all operations allows us to decouple the internal computational working set from the storage of results. The algorithmic core continues to operate on small, fixed-size blocks of A, S, and S', ensuring that the transient stack requirements for matrix generation and multiplication remain constant (approximately 2 kB) regardless of the security level. However, to achieve optimal performance by eliminating recomputation, Strategy C requires that the result matrices (B in KeyGen and B' in Encaps) are fully buffered in memory during computation.

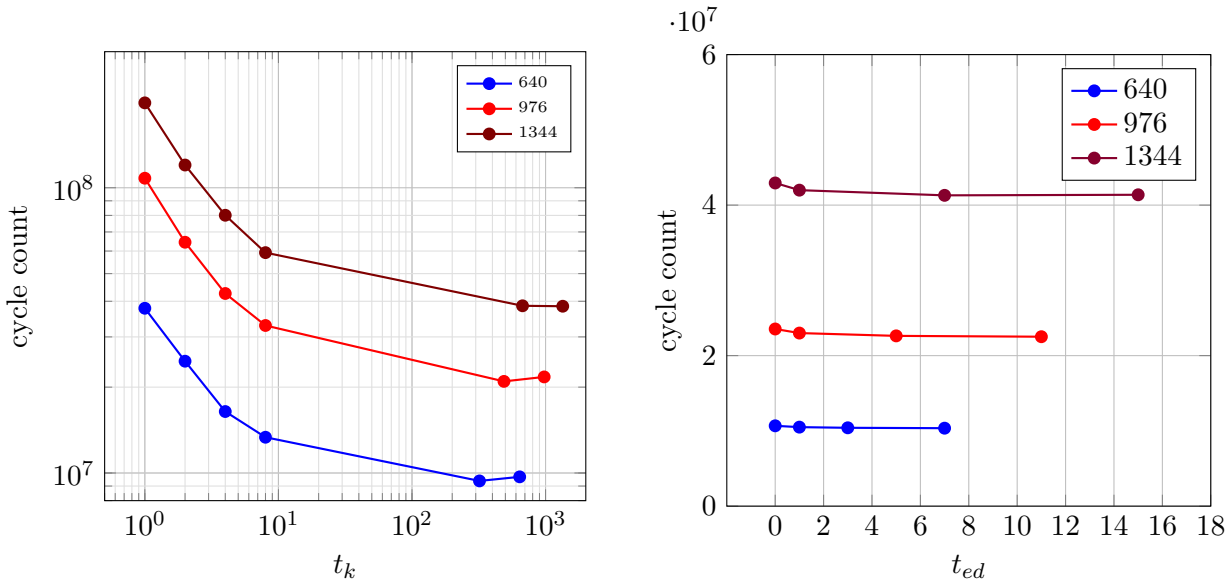
The total resultin memory consumption is listed in Table 5. Consequently, the usage scales linearly with the matrix dimension n. For instance, in KeyGen, the storage for B accounts for roughly 10 kB, 15 kB, and 21 kB for the three respective parameter sets.

Scheme	KeyGen Stack (bytes)	Encaps Stack (bytes)
FrodoKEM-640	12252	13208
FrodoKEM-976	17660	18624
FrodoKEM-1344	23584	24540

Table 5: Stack consumption of the KeyGen and Encaps operations under Strategy C.

10.2 Performance Analysis

Keygen Figure 3a presents the cycle counts for the KeyGen operation across varying buffer sizes t_k. The results highlight the significant impact of the recomputation overhead discussed in Section 9. For FrodoKEM-640, increasing the buffer size from $t_k = 1$ to $t_k = 640$ reduces the execution time by nearly 75%, dropping from $37.8 \cdot 10^6$ to $9.7 \cdot 10^6$ cycles, due to less frequent re-computation of the secret matrix S. Since the stack usage is constant regardless of t_k, choosing $t_k = n$ is optimal.



(a) KeyGen cycle counts (t_k)

(b) Encaps cycle counts (t_{ed})

Figure 3: Performance comparison of KeyGen and Encaps operations across parameter sets.

Encaps Conversely, Fig. 3b displays the performance of ENCAPS as a function of the horizontal tiling parameter t_{ed}. The cycle counts remain largely constant regardless of the tiling configuration. For FrodoKEM-640, the performance difference between the minimal tiling ($t_{ed} = 1$) and maximal tiling ($t_{ed} = 8$) is less than 3%. This indicates that under Strategy C, the cost of recomputing the small ephemeral secret S' is negligible compared to the dominant matrix arithmetic. Again, since the stack usage is constant regardless of t_{ed}, choosing $t_{ed} = \lceil \frac{2n}{rblock} \rceil$ is optimal.

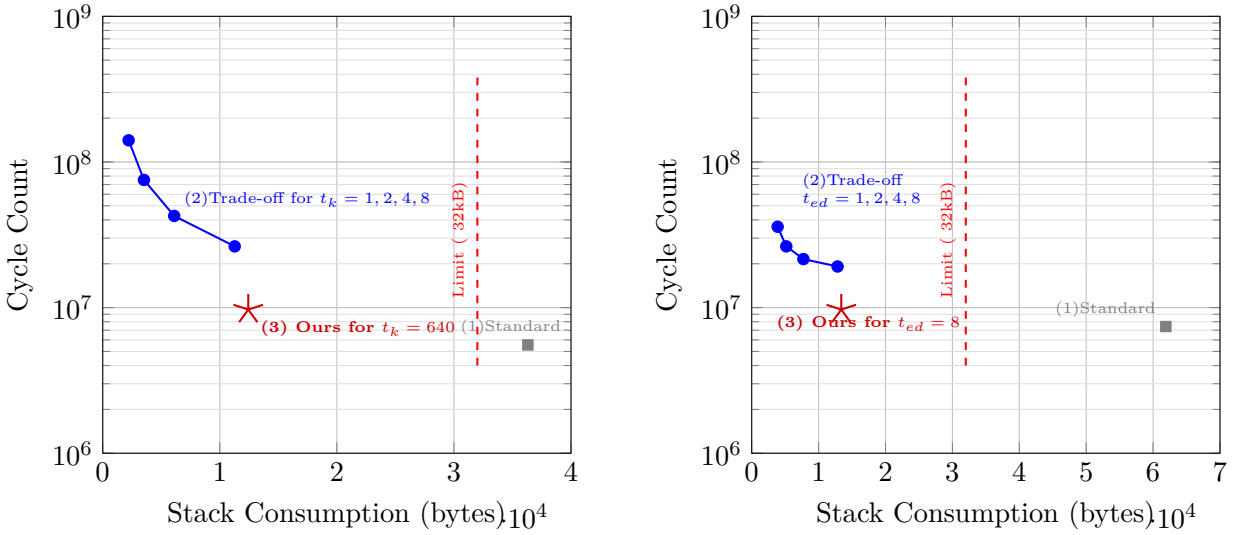
11 Comparison with Previous Works

To contextualize the performance benefits of the proposed blockwise secret generation, we compare three different implementations on the CV32E40P with custom extensions. All three implementations use tailored RISC-V assembly routines:

1. **Standard Implementation:** The reference implementation analogous to pqm4 [4], which generates matrices monolithically.
2. **Tiled Trade-off (A-only):** The implementation according to Bos et al. [12], which tiles only the public matrix A.
3. **Blockwise (A and S):** Our proposed implementation according to Strategy C with optimally chosen factors t_k and t_{ed}.

Key Generation Analysis. Figure 4a illustrates the performance landscape for FrodoKEM-640 KEYGEN. The standard implementation (1), while computationally efficient ($\approx 5.5 \times 10^6$ cycles), requires over 36 kB of stack memory. This exceeds the typical 32 kB SRAM limit of low-end embedded microcontrollers, effectively rendering the standard approach infeasible. The method by Bos et al. (2) enables execution on constrained devices but imposes a severe performance penalty. As shown by the blue curve, aggressive stack reduction (down to ≈ 2 kB) forces the cycle count

to spike to 140×10^6 cycles due to the repeated regeneration of the monolithic secret matrix S. In contrast, our proposed blockwise strategy C (point 3) somewhat mitigates the time–memory trade-off. As shown in the figure, we achieve the stack footprint of the least aggressive trade-off setting ($\approx 12\,\text{kB}$) while maintaining a cycle count of 9.7×10^6. This represents a 2.6× speed-up over the comparable low-memory configuration, and brings the performance within a factor of 1.75× of the standard implementation.



(a) Key Generation

(b) Encapsulation

Figure 4: **Cycle count vs. stack usage for FrodoKEM-640 with modifications.**

Encapsulation. A similar trend is observed for Encaps, detailed in Figure 4b. The standard implementation requires an excessive 62 kB of stack, which is double the available capacity of many MCUs. The trade-off implementation (2) allows execution with a stack size of $\approx 4\,\text{kB}$, but requires 36×10^6 cycles to run. Even if the stack budget is relaxed to $\approx 22\,\text{kB}$, the cycle count remains at 19×10^6. Our approach (3) executes in 10.4×10^6 cycles with a stack usage of roughly $\approx 13\,\text{kB}$. This offers a 1.9× speedup over the equivalent low-memory baseline. Notably, our result is only 1.4× slower than the hypothetical performance of the standard implementation, successfully decoupling memory constraints from algorithmic complexity.

Our contribution extends their architectural work by restructuring the algorithm to match the granularity of the hardware. The proposed blockwise flow operates on compact 168-byte blocks that fit entirely within small, low-latency stack buffers. This illustrates that hardware acceleration for lattice-based cryptography reaches its full potential only when paired with memory-aware algorithms that break long dependency chains, transforming raw accelerator power into sustained system-level performance.

12 Conclusion

In this work, we introduced a blockwise secret generation mechanism for FrodoKEM designed to address the prohibitive stack requirements of the standard specification on embedded devices. We provided a formal reduction showing that this modification preserves IND-CCA2 security in

the Random Oracle Model, ensuring the scheme remains as conservative as the reference design. Experimental evaluation on a 32-bit RISC-V core with custom cryptographic extensions demonstrates that this approach maintains a constant transient stack usage (approximately 2–3 kB) across all parameter sets, while allowing for a total memory footprint of roughly $12 - 24$ kB for FrodoKEM in performance-optimized configurations. This design effectively decouples the stack footprint from execution time. Consequently, it allows system designers to shift the focus towards strategies that optimize the accumulation of partial results, rendering FrodoKEM a more viable candidate for resource-constrained hardware.

Acknowledgments

This work was conducted as part of the author's Master's thesis at the Technical University of Munich (TUM). I would like to express my sincere gratitude to my supervisor, Patrick Karl, for his guidance, valuable insights, and support throughout the course of this research.

References

[1] Joppe Bos et al. "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, Oct. 24, 2016, pp. 1006–1018. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978425. URL: https://dl.acm.org/doi/10.1145/2976749.2978425 (visited on 11/21/2025).

[2] Erdem Alkım et al. "FrodoKEM Learning With Errors Key Encapsulation Algorithm Specifications And Supporting Documentation". In: 2019. URL: https://api.semanticscholar.org/CorpusID:218672966.

[3] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Journal of the ACM* 56.6 (Sept. 2009), pp. 1–40. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/1568318.1568324. URL: https://dl.acm.org/doi/10.1145/1568318.1568324 (visited on 11/17/2025).

[4] Matthias J Kannwischer et al. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. URL: https://github.com/mupq/pqm4.

[5] Joppe W Bos et al. "Fly, you fool! Faster Frodo for the ARM Cortex-M4". In: *Cryptology ePrint Archive* (2018). URL: https://eprint.iacr.org/2018/1116.pdf.

[6] James Howe et al. "Standard Lattice-Based Key Encapsulation on Embedded Devices". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Aug. 16, 2018), pp. 372–393. ISSN: 2569-2925. DOI: 10.46586/tches.v2018.i3.372-393. URL: https://tches.iacr.org/index.php/TCHES/article/view/7279 (visited on 11/26/2025).

[7] James Howe et al. "Exploring Parallelism to Improve the Performance of FrodoKEM in Hardware". In: *Journal of Cryptographic Engineering* 11.4 (Nov. 2021), pp. 317–327. ISSN: 2190-8508, 2190-8516. DOI: 10.1007/s13389-021-00258-7. URL: https://link.springer.com/10.1007/s13389-021-00258-7 (visited on 11/17/2025).

[8] Gökçe Düzyol, Muhammed Said Gündoğan, and Atakan Arslan. "Can FrodoKEM Run in a Millisecond? FPGA Says Yes!" In: *Cryptology ePrint Archive* (2025). URL: https://eprint.iacr.org/2025/1312.pdf.

[9] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Aug. 9, 2019), pp. 17–61. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i4.17-61. arXiv: 1910.07557[cs]. URL: http://arxiv.org/abs/1910.07557 (visited on 11/26/2025).

[10] Viet B. Dang et al. "Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign". In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019 International Conference on Field-Programmable Technology (ICFPT). Tianjin, China: IEEE, Dec. 2019, pp. 206–214. ISBN: 978-1-7281-2943-3. DOI: 10.1109/ICFPT47387.2019.00032. URL: https://ieeexplore.ieee.org/document/8977901/ (visited on 11/26/2025).

[11] Patrick Karl, Tim Fritzmann, and Georg Sigl. "Hardware Accelerated FrodoKEM on RISC-V". In: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS). Prague, Czech Republic: IEEE, Apr. 6, 2022, pp. 154–159. ISBN: 978-1-6654-9431-1. DOI: 10.1109/DDECS54261.2022.9770148. URL: https://ieeexplore.ieee.org/document/9770148/ (visited on 11/17/2025).

[12] Joppe W. Bos et al. "Enabling FrodoKEM on Embedded Devices". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (June 9, 2023), pp. 74–96. ISSN: 2569-2925. DOI: 10.46586/tches.v2023.i3.74-96. URL: https://tches.iacr.org/index.php/TCHES/article/view/10957 (visited on 11/17/2025).

[13] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Journal of Cryptology* 26.1 (Jan. 2013), pp. 80–101. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/s00145-011-9114-1. URL: http://link.springer.com/10.1007/s00145-011-9114-1 (visited on 11/17/2025).